Strings

- Strings are actually one-dimensional array of characters terminated by a null character '\0'.
- Thus a null-terminated string contains the characters that comprise the string followed by a null.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```c
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization, then you can write the above statement as follows:
```c
char greeting[] = "Hello";
```

Actually, you do not place the null character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string:

```c
#include <stdio.h>
main ()
{
    char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
    printf("Greeting message: %s\n", greeting);
}
```

When the above code is compiled and executed, it produces the following result:

```
Greeting message: Hello
```

C supports a wide range of functions that manipulate null-terminated strings:

1) `char *strcpy(char *s1, char *s2);`
   Copies string s2 into string s1.

2) `char *strcat(char *s1, char *s2);`
   Concatenates string s2 onto the end of string s1.
3) size_t strlen(char *s1);
   Returns the length of string s1.

4) int strcmp(char *s1, char *s2);
   Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.

5) char *strchr(char *s1, char ch);
   Returns a pointer to the first occurrence of character ch in string s1.

6) char *strstr(char *s1, char *s2);
   Returns a pointer to the first occurrence of string s2 in string s1.

7) char *strlwr(char *s1)
   Converts string pointed by s1 to lowercase.

8) char *strupr(char *s1)
   Converts string pointed by s1 to uppercase.

The following example uses some of the above-mentioned functions:
#include <stdio.h>
#include <string.h>
int main ()
{
    char str1[12] = "Hello";
    char str2[12] = "World";
    char str3[12];
    int len ;

    // copy str1 into str3
    strcpy(str3, str1);
    printf("strcpy( str3, str1) : %s\n", str3 );

    // concatenates str1 and str2
    strcat( str1, str2);
    printf("strcat( str1, str2): %s\n", str1 );

    // total length of str1 after concatenation
    len = strlen(str1);
    printf("strlen(str1) : %d\n", len );
    return 0;
}

When the above code is compiled and executed, it produces the following result:
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
Pointers and Strings

Suppose we wish to store "Hello". We may either store it in a string or we may ask the C compiler to store it at some location in memory and assign the address of the string in a char pointer. This is shown below:

```c
char str[] = "Hello";
char *p = "Hello";
```

There is a subtle difference in usage of these two forms. For example, we cannot assign a string to another, whereas, we can assign a char pointer to another char pointer. This is shown in the following program.

```c
main()
{
    char str1[] = "Hello";
    char str2[10];
    char *s = "Good Morning";
    char *q;
    str2 = str1; /* error */
    q = s;        /* works */
}
```

Also, once a string has been defined it cannot be initialized to another set of characters. Unlike strings, such an operation is perfectly valid with char pointers.

```c
main()
{
    char str1[] = "Hello";
    char *p = "Hello";
    str1 = "Bye";    /* error */
    p = "Bye";      /* works */
}
```

**Write a program to find length of string using pointer.**

```c
int strlength ( char *s )
{
    int length = 0;
    while ( *s != \0 )
    {
        length++;
        s++;
    }
    return ( length ) ;
}
main()
{
    char str[80];
    gets(str);
    printf("\nLength = %d",strlength(str));
}
```
Write a program to copy one string to another and return copied string.
char * strcopy ( char *s2, char *s1 )
{
    char *t = s2;
    while ( *s1 != '\0' )
    {
        *s2 = *s1;
        s1++ ;
        s2++ ;
    }
    *s2 = '\0';
    return t ;
}
main()
{
    char source[80], target[80];
    gets(source);
    puts(strcopy(target, source));
}

Array of Pointers to Strings
As we know, a pointer variable always contains an address. Therefore, if we construct
an array of pointers it would contain a number of addresses.

Example :
char *names[] = {
    "akshay",
    "parag",
    "raman",
    "srinivas",
    "gopal",
    "rajesh"
} ;
In this declaration names[] is an array of pointers. It contains base addresses of
respective names. That is, base address of “akshay” is stored in names[0], base address
of “parag” is stored in names[1] and so on. This is depicted in Figure.
In the two-dimensional array of characters, the strings occupied 60 bytes. As against this, in array of pointers, the strings occupy only 41 bytes—a net saving of 19 bytes.

```c
main()
{
    char *city[]={"Pune","Mumbai","Satara","Solapur","Delhi");
    char *temp;
    printf("Original : %s\t%s",city[2],city[3]);
    temp=city[2];
    city[2]=city[3];
    city[3]=temp;
    printf("New : %s\t%s",city[2],city[3]);
}
Original : Satara    Solapur
New:Solapur  Satara
```

**Arrays of Strings**

It is not uncommon in programming to use an array of strings. For example, the input processor to a database may verify user commands against an array of valid commands. To create an array of strings, use a two-dimensional character array. The size of the left dimension determines the number of strings, and the size of the right dimension specifies the maximum length of each string.

```c
char str_array[30][80];
```

For example, the following statement calls `gets()` with the third string in `str_array`.

```c
gets(str_array[2]);
```

The preceding statement is functionally equivalent to

```c
gets(&str_array[2][0]);
```

but the first of the two forms is much more common in professionally written C code.

The following declares an array of 3 strings, each with a maximum length of 10 characters:

```c
char city[3][10]={"Pune","Mumbai","Satara"};
```

It is easy to access an individual string: You simply specify only the left index.
Program to accept and display n names.

```c
#include<stdio.h>
#include<string.h>
main()
{
    char names[10][20];
    int i,n;
    printf("Enter how many names : ");
    scanf("%d",&n);
    /*Accept n names*/
    for(i=0;i<n;i++)
    {
        printf("Enter name %d",i);
        gets(names[i]);
    }
    /*Displaying n names*/
    for(i=0;i<n;i++)
    {
        puts(names[i]);
    }
}
```